

Mache: No-Loss Trace Compaction

A. Dain Samples

Computer Science Division-EECS
University of California at Berkeley
Berkeley, CA 94720

September 15, 1988

Abstract

Execution traces can be significantly compressed using their referencing locality. A simple observation leads to a technique capable of compressing execution traces by an order of magnitude; instruction-only traces are compressed by two orders of magnitude. This technique is unlike previously reported trace compression techniques in that it compresses without loss of information and, therefore, does not affect trace-driven simulation time or accuracy.

Report Documentation Page				Form Approved OMB No. 0704-0188	
Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.					
1. REPORT DATE 15 SEP 1988		2. REPORT TYPE		3. DATES COVERED 00-00-1988 to 00-00-1988	
4. TITLE AND SUBTITLE Mache: No-Loss Trace Compaction				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of California at Berkeley, Department of Electrical Engineering and Computer Sciences, Berkeley, CA, 94720				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT Execution traces can be significantly compressed using their referencing locality. A simple observation leads to a technique capable of compressing execution traces by an order of magnitude; instruction-only traces are compressed by two orders of magnitude. This technique is unlike previously reported trace compression techniques in that it compresses without loss of information and, therefore, does not affect trace-driven simulation time or accuracy.					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT Same as Report (SAR)	18. NUMBER OF PAGES 24	19a. NAME OF RESPONSIBLE PERSON
a. REPORT unclassified	b. ABSTRACT unclassified	c. THIS PAGE unclassified			

Mache: No-Loss Trace Compaction*

A. Dain Samples[†]

September 15, 1988

Abstract

Execution traces can be significantly compressed using their referencing locality. A simple observation leads to a technique capable of compressing execution traces by an order of magnitude; instruction-only traces are compressed by two orders of magnitude. This technique is unlike previously reported trace compression techniques in that it compresses without loss of information and, therefore, does not affect trace-driven simulation time or accuracy.

1 Introduction

Collecting and storing execution traces produces huge amounts of data that can tax the largest disk system (or, more precisely, other users of that disk system). Traces on the order of tens of millions of memory references requiring anywhere from forty to fifty megabytes are not uncommon.

In section 2, I discuss a basic method for reducing the space required to store an execution trace significantly. The technique has the following properties.

*This research was sponsored by the Defence Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292.

[†]Supported in part by an AT&T Bell Laboratories Scholarship.

- No information is lost.
- Compaction is in direct proportion to the locality of the memory references in the trace.
- It is a one-pass algorithm with running time $O(n)$.

Section 4 examines why the technique works so well, and looks to see if any improvements are possible. While improvements are possible, the results are not sufficiently better to warrant the additional complexity.

2 Method

I assume that the memory traces of interest are similar to DineroIII input format [2]. Each memory reference is a pair *(label address)*: the label identifies the type of reference and the address is an unsigned integer. In the memory traces used in this paper, there are three labels: a label for an instruction fetch, a label for data read, and a label for data write.

The technique described here, called *Maching* (rhymes with ‘caching’), loses no information and compresses general memory reference traces (instruction and data) by factors of ten to twenty times and instruction address traces by factors of thirty to two hundred times depending upon locality.

The basic technique is in Figure 1: when a *(label address)* pair is read, the difference $\delta = \text{cache}_{\text{label}} - \text{address}$ is computed between this address and the last address for this label. If this difference is small enough, it is emitted in the low-order 14 bits of the codeword; otherwise the full address is emitted preceded by a ‘miss’ indicator.

The routines *emitByte*, *emit2bytes*, and *emit4bytes* pass their argument of the indicated size to a dynamic, one-pass compression scheme which takes advantage of repeating patterns in the sequence of bytes. I have used the Lempel-Ziv compression algorithm [9,10,11] implemented as the *compress* program un-

```

encode(label,address) {
     $\delta \leftarrow \text{cache}[\text{label}] - \text{address};$ 
    if ( $|\delta| \geq \text{threshold}$ ) {
        /* A 'miss'; emit the full address */
        codeByte  $\leftarrow (\text{missFlag} \ll 6) \mid \text{label};$ 
        emitByte(codeByte);
        emit4bytes(address);
    }
    else {
        /* A 'hit'; emit the shorter difference value */
        codeWord  $\leftarrow (\text{label} \ll 14) \mid \delta;$ 
        emit2bytes(codeWord);
    }
    cache[label]  $\leftarrow \text{address};$ 
}

```

10

Figure 1: The inner loop

der UNIX¹. The Lempel-Ziv technique (hereafter referred to as LZ) is extraordinarily capable of detecting and exploiting the memory referencing patterns exposed by the differencing of the execution trace.

2.1 Traces

We have used several different execution traces of programs running to test the efficacy of *Mache* (see Table 1 for quantitative descriptions). COMP is a trace² of *compress* compressing a file of C source code (a concatenation of all the source code for a version of *Mache*). This file of source code was 90,872 bytes long and compressed to 38006 bytes. The *compress* program has a very tight inner loop for very high locality in its instruction references.

CC1 is a trace of the Gnu C compiler compiling the 303 line file *genemit.c*, which is one of the source files for the Gnu C compiler. The file contains 276

¹UNIX is a trademark of AT&T Bell Laboratories.

²All traces reported in this paper were taken on a Sun workstation using the 68020.

name	# references	bytes	command
COMP	12,437,454	62,187,270	compress <sources.c >temp.Z
CC1	13,711,558	68,557,790	gcc genemit.c
VAX	13,898,315	69,491,575	vaxima <vax.in
TROFF	13,092,334	65,461,670	troff -t -Tip -rv1 paper.short
LATEX	27,221,840	136,109,200	latex todo
SCR-N	16,368,894	81,844,470	scrunch <scrunch.c
SCR-R	16,651,110	83,255,550	xscrunch <scrunch.c

Table 1: Summary of traces used

lines, is 4,664 bytes long sans comments, and has nine function definitions.

VAX is a trace of a version of Vaxima (a Macsyma-like symbolic processor) solving eight simultaneous equations in nine variables. The trace includes the reading of the input and the displaying of the answer. It was included since Vaxima is implemented in LISP, a language often charged with the sin of producing non-local memory referencing behavior.

TROFF is a trace of the troff typesetting system working on a short paper. The file is 7705 bytes long, and produces 9768 bytes of output. This trace is included because of the nature of the troff source code. Troff was more or less transliterated into C from an assembly language implementation that made heavy use of jump tables [8]. The resulting C code is peppered with *switch* statements and non-structured gotos, and has bad locality.

LATEX is a trace of the \LaTeX text processor constructing a “To Do List” form. The resulting typeset page of output has very little text and a lot of straight lines. \LaTeX is a macro package running on a Common \TeX version of Knuth’s text processing system [3].

SCR-N and SCR-R are traces of a Huffman encoder called *scrunch* compressing a 1529 line, 41,796 byte file (its own source) that is compressed to 27,345 bytes. SCR-N is the trace of a version of *scrunch* produced directly by the Gnu C compiler. SCR-R is the trace of a version of *scrunch* that has been modified to improve the locality of the code. See reference 6 for more information on this research. The important point is that these two traces show the effect of the

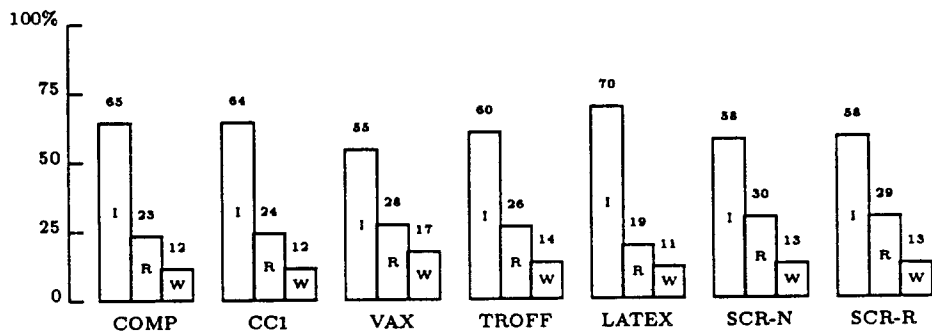


Figure 2: Breakdown of memory references in each trace

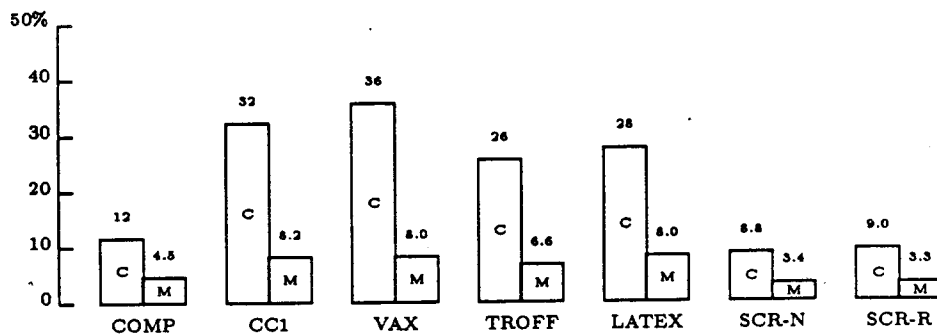


Figure 3: compress vs. Mache, percent of original trace

locality of an execution trace on *Mache*'s performance.

In Figure 2 we see that the traces are similar in their mix: 50-60% instruction references, 20-30% data reads and 15-25% data writes. The numbers for any one trace may not add to 100 due to rounding errors. In particular, SCR-N has slightly less than 30% reads, and slightly more than 58% instruction fetches, while SCR-R has more instruction fetches than SCR-N, but the same number of reads and writes.

3 Results

Figure 3 shows the results of compressing each trace by *compress* (C) and *Mache* (M). From these traces we see that *Mache* almost always creates a file at least

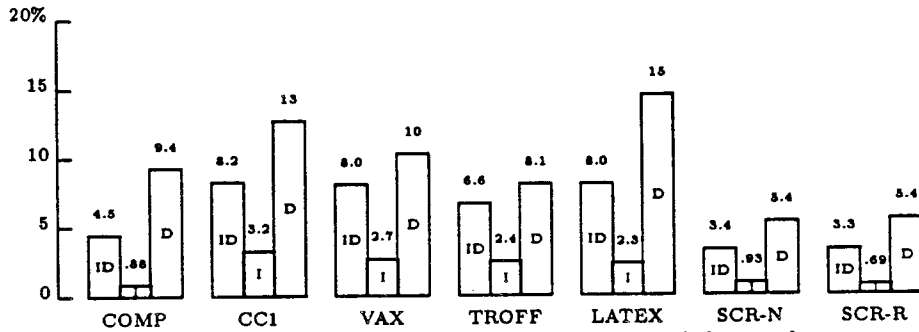


Figure 4: Compression of instruction-only and data-only traces

ten times smaller than the original trace file and over three times smaller than that produced by *compress*.

This scheme works exceptionally well for instruction-only traces. Figure 4 compares the compaction achieved on the full trace (ID), with the compaction achieved on only instruction references (I) and the data references (D). The percentages in the graph are relative to the sizes of the traces with all memory references not of interest thrown out. For example, the full trace COMP has 12,437,454 references, occupying 62,187,270 bytes in raw form. Of those references, 8,065,326 are instruction references which take 40,326,630 bytes in raw form, but only 0.88% of that (355,143 bytes) in compressed form. It is apparent that the bulk of the space savings for a complete trace is derived from the regularity in the instruction stream.

Traces SCR-N and SCR-R are of particular note since they show the sensitivity of the compression scheme to the locality of the trace. Both are traces of the same program working on the same data. SCR-N is a trace of the program as it is produced by the Gnu C compiler, and SCR-R is a trace of the program after its basic blocks and procedures were reorganized to improve the program's behavior in an instruction cache. In all other respects, the two versions of the program were the same. The difference in compression between the two traces is attributable solely to the differences in the locality of the memory references be-

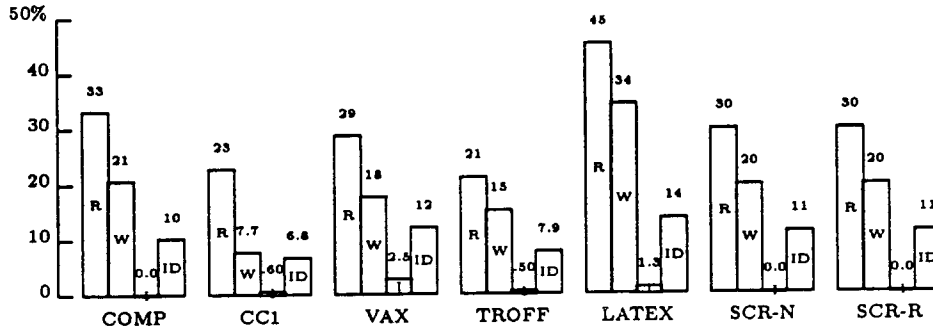


Figure 5: Miss rates

tween the two versions of the program. It is interesting to note that even though trace SCR-R contains more instructions fetched and executed than SCR-N (see Table 2), the reorganized program nevertheless executes faster and the trace compresses better due to the improved locality of the code.

In Figure 5 the 'miss rate' for each kind of reference for each trace is plotted. The miss rate is the percentage of references that require a full address to be emitted because the difference between the last and current address is too large to be encoded in a short code word. Each trace has four miss rates shown, one for each kind of memory reference (read, write, and instruction fetch) and the overall miss rate.

4 Why it works

There are four reasons why this scheme works: (1) encoding differences between 32-bit addresses often requires fewer bits than encoding the addresses, (2) the differences tend to cluster much more than the original addresses, (3) the stream of differences is much more regular than the stream of addresses, and (4) the LZ compression technique works solely by discovering and encoding common sequences of bytes in the source. These attributes can be seen in Figure 6. The two columns on the left are the first few memory references from the LATEX

original trace		difference trace		comments
2	2020	3,2	2020	first instruction ref
0	efffd64	3,0	efffd64	first data read, from stack
0	efffd66	0	2	second data read
2	2022	2	2	
2	2024	2	2	
2	2026	2	2	
2	2028	2	2	
2	202a	2	2	
2	202c	2	2	
2	202e	2	2	
2	2030	2	2	
2	2032	2	2	
1	40000	3,1	40000	first data write
1	40002	1	2	
2	2034	2	2	
1	efffd60	3,1	efbfd5e	a data write 'miss'
1	efffd62	1	2	
2	2036	2	2	
1	efffd5c	1	-6	
1	efffd5e	1	2	
2	2038	2	2	
1	efffd58	1	-6	
1	efffd5a	1	2	
2	203a	2	2	
2	203c	2	2	
2	203e	2	2	
1	efffd54	1	-6	
1	efffd56	1	2	
2	2066	2	28	
2	2096	2	30	
0	efffd54	0	-12	
0	efffd52	0	-2	
2	2040	2	-56	
2	2042	2	2	
2	2044	2	2	
1	efffd54	1	-2	
1	efffd56	1	2	
2	2098	2	54	
2	209a	2	2	
2	209c	2	2	
2	2d898	3,2	2b7fc	an instruction fetch 'miss'
2	2d89a	2	2	

Figure 6: Sample trace from LATEX
8

trace, with the resulting stream of differences in the middle column. (The notation "3,2" means that a reference with label 2 missed.) Differencing results in more redundancy, the numbers required to represent the differences are much smaller, and there are patterns in the difference stream that are not apparent in the original. (All numbers are in hexadecimal).

Finally, the distribution of byte-values in the trace file has little to do with the final compression. For example, the number of bytes in the raw trace file CC1 is 68,557,790, and the number of bytes after differencing is 27,830,296. The entropies of the two traces are 5.40 and 3.55 respectively, implying that a Huffman encoding of the raw file would be about 46.28 megabytes, and of the difference file about 12.35 megabytes. However, these are nowhere near the 5,649,247 bytes produced by the combination of differencing and compressing.

The best compression is not achieved by looking at individual byte values and their distributions, but by the discovery and encoding of common sub-sequences exposed by differencing. 'Bit twiddling' variations based on this basic method result in negligible differences.

5 Refinements and Variations

I have tried several variations of the scheme presented so far to compress trace files even further. In actual fact, most of these 'variations' were tried first, and led to the conclusion that the simpler scheme presented above was the better one. I'll recapitulate the ontogeny of the technique, and then report on the results of the variations tried along the way.

The original scheme came from thinking about caches (and hence the name *Mache*). The first attempt simply coded three caches, one for instruction fetches, one for data reads, and another for data writes. Then, whenever a (*label address*) pair was read, *cache_{label}* was examined for a slot *i* which contained the address. If such a slot were found (a 'hit'), a 16-bit word was created which encoded

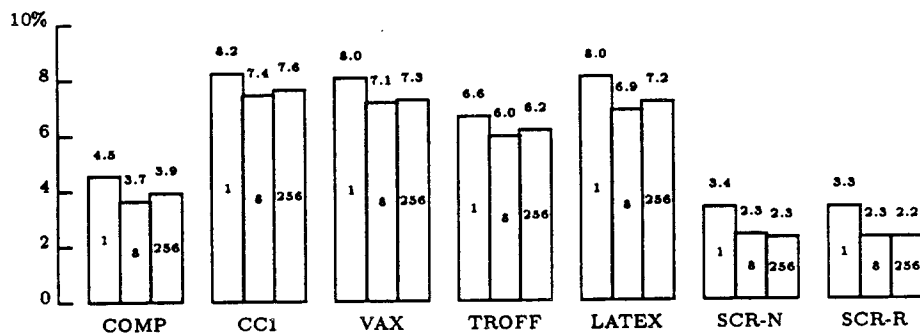


Figure 7: Maching with 1/8/256-slot caches

the label (and hence the cache), the slot number, and the offset within the slot. If it were not found (a 'miss') a header byte was emitted that encoded the miss for this label, followed by the emission of the four byte address; this stream of encoded cache-hit records is then compressed with an LZ algorithm. I tried several of the traditional replacement algorithms for caches and, not surprisingly, LRU worked best.

With a little thought came the realization that if the cache were made into a LIFO stack with LRU replacement, - i.e. when a slot was hit, it became slot zero and all the slots in between were moved down - then almost all hits would be to slot zero. This would increase the amount of redundancy in the resulting cache-hit stream. This modification produced significant improvements, and was the method of choice for some time. Note that the method presented in section 2 is this method using a cache with one slot.

But what is the optimal number of slots to produce the best compression? Figure 7 shows each of the traces *Mached* with different sizes of caches: the original 1-slot cache, an 8-slot cache, and a 256-slot cache. Apparently, it makes little difference.

To test this further, I tried compressing a single trace using different numbers of cache slots. Figure 8 is the result of *Maching* the CC1 trace using one, two, four, six, eight, ten, sixteen, thirty-two and 256 slots in the caches.

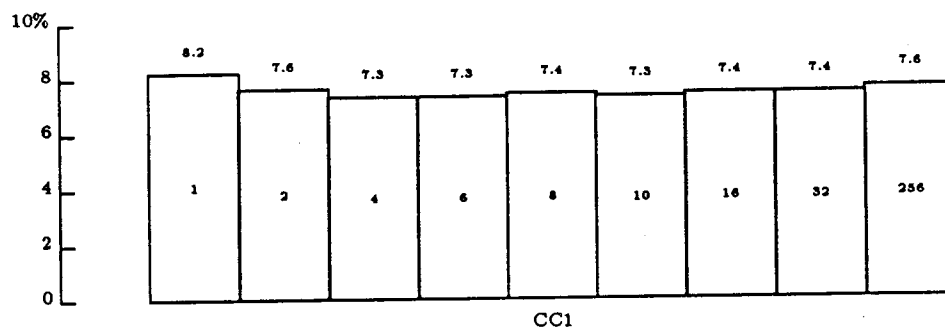


Figure 8: Matching CC1 with 1/2/4/6/8/10/16/32/256-slot caches

Even though there appears to be a minimum somewhere between six and ten slots³ there is apparently not enough improvement over the 1-slot scheme (which is exactly equivalent to the 'differencing' method presented in section 2) to warrant the additional complexity.

There were a few other variations tried that "seemed like a good idea at the time" but which produced either worse results or negligibly different results. I will not report details on them here, but just give the bottom line. For example, the thought occurred to me that if each instruction word in memory could be given its own data cache, the memory reference patterns associated with each instruction would readily become apparent. Since not every word can be given such a cache, we approximated it by hashing the address of an instruction word into a set of 256 caches. However, either the number of caches was insufficient, or the patterns were not uncovered, because the results were surprisingly disappointing.

Another scheme used an eight-bit byte to encode hits and misses rather than the 16 bit words used in the experiments reported elsewhere in this paper. The results were negligibly different, and not always better. When using an eight-bit encoding, any differences larger than about 40 require a 32-bit word to record

³The 'bump' at eight slots in Figure 7 is probably attributable to vagaries of the LZ compression scheme; in particular, when the hash table in LZ fills up and the compression ratio begins to fall, LZ clears everything and starts over afresh.

the 'miss'. Such misses are now much more frequent, and account for a larger percentage of the resulting compression file.

And, finally, I split the trace out into three separate instruction, data-read, and data-write streams. This required the existence of a fourth stream to record the order of the reads, writes, and instruction fetches. The reasoning behind this said that if instruction streams could be reduced two orders of magnitude when taken by themselves, perhaps that compression combined with increased compression of the data-read and data-write streams would result in a better compression overall. Alas and alack, the fourth stream (the order stream) turns out to be pretty random already, and did not compress sufficiently to make everything come out smaller on the whole. While it was true that the sum of the sizes of the compressed instruction, data-read, and data-write streams was smaller than the size of the original compressed trace file, the size of the compressed order file was too large, and made the total file larger. Besides which, this scheme was far too complicated to recommend in good conscience.

One variation I *haven't* tried is run-length encoding: that is, after differencing, encoding the number of times the same cache-hit value occurs sequentially. This run-length encoded stream can then be passed to the LZ compression scheme. While this *may* produce impressive results for instruction-only trace files, I suspect that it will do nothing for data-only traces, and will have marginal benefit for full traces. The results presented in the next section support this conjecture.

5.1 Other Applications

This scheme should not be relegated to compressing only trace data. Any stream of data that contains first-order difference patterns can benefit from differencing followed by LZ encoding. For example, image bit maps, particularly those using multiple bits per pixel (e.g. color images), and analog-to-digital data that changes relatively slowly over time would be very susceptible to compression by

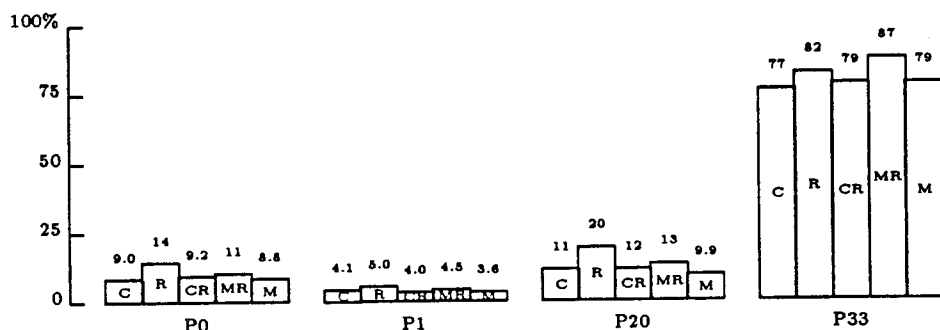


Figure 9: Mache vs. compress vs. run length encoding; percent of original file

this technique.

To show this, we have compressed four color bit-images⁴: P0 is a picture of the Mandelbrot set [4], and P1, P20, and P33 are magnified details of this set. The size of each image is 722,532 bytes: eight bits of color information per pixel for an 850 by 850 image. In terms of visual complexity, P1 is the least complex, and P33 is the most complex, with P20 falling somewhere in between. This is reflected in the pictures' respective compressibility. Figure 9 shows that, with the exception of P33, *Mache* (M) obtains the better compression over *compress* (C), a run length encoding (R), and a *compress* of the run length encoding (CR). *Maching* the original picture also does better than *Maching* the run length encoded version of the picture (MR).

5.2 Performance

It should be fairly obvious from Figure 1 that the overhead required to do the differencing is minimal: keep the last address and pass to the LZ compression routine the difference between the last and the current address. The fact that even with this overhead *Mache* is often faster than *compress* may be surprising. Since differencing produces a file that is approximately 40% of the size of the original, the LZ routine has fewer bytes as input. Given that a good portion

⁴Black and white versions of the pictures appear in an appendix to this paper.

of the time in LZ compression is spent hashing, shifting, *anding*, and *oring* variable length bit strings, the overhead of a subtraction and a compare or two is more than compensated by the work saved in the LZ compression routine. However, LZ works much faster when decompressing a file, so the overhead in *Mache* of caching and adding addresses represents a greater proportion of the execution time when it is decompressing. Nevertheless, the performance figures are very close. For example, on a MIPS processor, it took 408 CPU seconds to *compress* the COMP trace, but only 315 seconds to *Mache* it. To *uncompress* takes 215 CPU seconds, but to *unMache* it takes 229 seconds. (For comparison, it takes the *cmp* program over 170 seconds to compare the uncompressed 65Mb COMP trace with a copy of itself.) *Mache* compressed a smaller execution trace from 691,460 bytes to 67,583 bytes in 3.3 CPU seconds, while *compress* took it down to 235,971 bytes in 4.7 seconds. *Mache* took 2.5 seconds to recover the trace, while *compress* required 3.3 seconds.

Bottom line: *Mache* is fast.

6 Related Work

Most of the related work in trace compaction has concentrated on decreasing the amount of time required to do a simulation (of caches, virtual memory, etc.) as well as the amount of space required to store the cache. These compression schemes have often thrown away information that is relatively unimportant to the intended simulations. The technique reported in this paper has concentrated solely on space savings. If decreasing simulation times are important, then one of the following techniques may prove useful.

Alan Smith [7] reports on two methods for reducing the size of address traces. The first method, stack deletion, deletes the top k levels of the LRU stack. Information loss introduces an error rate in paging simulations on the order of less than 1% while reducing the trace size from 25-95%. The second

method, snapshot, reduces trace sizes by a factor of 5 to 100 with an introduced error rate similar to that for stack deletion. However, these error rates appear to be valid only for paging or cache studies using full associative placement, and do not apply to set-associative caches.

Thomas Puzak [5] reported a method he called *tape stripping* in which a direct-mapped cache with a fixed block size is simulated. Only the misses and run-length of the hits are recorded from the simulation, resulting in compaction on the order of 90-95%. He proves that there is no error introduced if the stripped trace is used to simulate caches with more sets and the same block size. He also shows that if different block sizes are used, the introduced relative error is small ($\leq 1\%$) and becomes negligible as the block size becomes large. He suggests using different traces that have been stripped using the block size of interest. Of course, this requires keeping the original raw trace around, in which case *Mache* will be useful.

Anant Agarwal [1] studied cache behavior in multi-processor systems with hierarchical caches. His compaction technique can produce reductions of one to two orders of magnitude, and introduces simulation errors on the order of 10-15% in measured miss ratios due to information loss.

7 Conclusions

Mache is a simple and surprisingly effective technique for compacting execution traces. It uses differencing of memory references to expose patterns in program execution that are not immediately available in the raw trace. Using a compaction scheme that looks for and encodes long common sequences of bytes results in compressed trace files less than 10% of original size for complete traces, and on the order of 0.5% the size of the original trace for instruction-only traces.

Acknowledgements: The final form of this paper been improved considerably by constructive input from several people. Thanks to Paul Hilfinger,

David Wood, Charlie Farnum and especially Alan Smith for their comments.
Any problems that remain are mine, and are probably the result of my not
paying closer attention to their suggestions.

8 Bibliography

1. AGARWAL, A. Trace Compaction Using Cache Filtering with Blocking. draft, 1987.
2. HILL, M. D. DineroIII Cache Simulator. University of California, Berkeley, UNIX Programmer's Manual, August 1985.
3. KNUTH, D. *The TeXbook*. Addison Wesley, Reading, MA, 1984.
4. MANDELBROT, B. B. *The fractal geometry of nature*. W. H. Freeman and Company, San Francisco, CA, 1983.
5. PUZAK, T. R. Analysis of Cache Replacement-Algorithms. University of Massachusetts, PhD Dissertation, February 1985.
6. SAMPLES, A. D. Code Reorganization for Instruction Caches. Computer Science Division, EECS, University of California, Berkeley, Technical Report UCB/CSD 88/447, 1988.
7. SMITH, A. J. Two Methods for the Efficient Analysis of Memory Address Trace Data. *IEEE Transactions on Software Engineering SE-31* (January 1977).
8. WALL, D. Register Windows vs. Register Allocation. *SIGPLAN '88 Conference on Programming Language Design and Implementation 237* (June 22-24, 1988), 67-78.
9. WELCH, T. A. A Technique for High Performance Data Compression. *IEEE Computer 176* (June 1984), 8-19.
10. ZIV, J. AND LEMPEL, A. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory 23* (1976), 75-81.
11. ZIV, J. AND LEMPEL, A. Compression of Individual Sequences via Variable-rate Coding. *IEEE Transactions on Information Theory 24* (1978), 530-536.

The complex-plane coordinates of the lower left-hand corner of each picture can be found in the box in the upper right. 'Size' refers to the number of pixels to a side of the image on the original color display.

Pict = p000

Real = -2.0000000000000000

Imag = -1.2500000000000000

Size = 2.5000000000000000

Size = 850

Current ColorMap = bw120

Canvas Real = -2.0000000000000000

Canvas Imag = -1.2500000000000000

Canvas Size = 2.5000000000000000

Done

Load	Disp	Inv.	Note	Quit
p000	p10	p20	p1	p2
p1	p2	p3	p4	p5
p6	p7	p8	p9	p10
p11	p12	p13	p14	p15
p16	p17	p18	p19	p20

Rotate Speed:

Rotate Speed:

Default
1/5 Rotate
Full Rotate
Invert
Map Rotate

ColorMaps

bw000	bw001	bw002
bw003	bw004	bw005
bw006	bw007	bw008
bw009	bw010	bw011
bw012	bw013	bw014
bw015	bw016	bw017
bw018	bw019	bw020
bw021	bw022	bw023
bw024	bw025	bw026
bw027	bw028	bw029
bw030	bw031	bw032
bw033	bw034	bw035
bw036	bw037	bw038
bw039	bw040	bw041
bw042	bw043	bw044
bw045	bw046	bw047
bw048	bw049	bw050
bw051	bw052	bw053
bw054	bw055	bw056
bw057	bw058	bw059
bw060	bw061	bw062
bw063	bw064	bw065
bw066	bw067	bw068
bw069	bw070	bw071
bw072	bw073	bw074
bw075	bw076	bw077
bw078	bw079	bw080
bw081	bw082	bw083
bw084	bw085	bw086
bw087	bw088	bw089
bw090	bw091	bw092
bw093	bw094	bw095
bw096	bw097	bw098
bw099	bw100	bw101
bw102	bw103	bw104
bw105	bw106	bw107
bw108	bw109	bw110
bw111	bw112	bw113
bw114	bw115	bw116
bw117	bw118	bw119
bw120	bw121	bw122
bw123	bw124	bw125
bw126	bw127	bw128
bw129	bw130	bw131
bw132	bw133	bw134
bw135	bw136	bw137
bw138	bw139	bw140
bw141	bw142	bw143
bw144	bw145	bw146
bw147	bw148	bw149
bw150	bw151	bw152
bw153	bw154	bw155
bw156	bw157	bw158
bw159	bw160	bw161
bw162	bw163	bw164
bw165	bw166	bw167
bw168	bw169	bw170
bw171	bw172	bw173
bw174	bw175	bw176
bw177	bw178	bw179
bw180	bw181	bw182
bw183	bw184	bw185
bw186	bw187	bw188
bw189	bw190	bw191
bw192	bw193	bw194
bw195	bw196	bw197
bw198	bw199	bw200
bw201	bw202	bw203
bw204	bw205	bw206
bw207	bw208	bw209
bw210	bw211	bw212
bw213	bw214	bw215
bw216	bw217	bw218
bw219	bw220	bw221
bw222	bw223	bw224
bw225	bw226	bw227
bw228	bw229	bw230
bw231	bw232	bw233
bw234	bw235	bw236
bw237	bw238	bw239
bw240	bw241	bw242
bw243	bw244	bw245
bw246	bw247	bw248
bw249	bw250	bw251
bw252	bw253	bw254
bw255	bw256	bw257
bw258	bw259	bw260
bw261	bw262	bw263
bw264	bw265	bw266
bw267	bw268	bw269
bw270	bw271	bw272
bw273	bw274	bw275
bw276	bw277	bw278
bw279	bw280	bw281
bw282	bw283	bw284
bw285	bw286	bw287
bw288	bw289	bw290
bw291	bw292	bw293
bw294	bw295	bw296
bw297	bw298	bw299
bw300	bw301	bw302
bw303	bw304	bw305
bw306	bw307	bw308
bw309	bw310	bw311
bw312	bw313	bw314
bw315	bw316	bw317
bw318	bw319	bw320
bw321	bw322	bw323
bw324	bw325	bw326
bw327	bw328	bw329
bw330	bw331	bw332
bw333	bw334	bw335
bw336	bw337	bw338
bw339	bw340	bw341
bw342	bw343	bw344
bw345	bw346	bw347
bw348	bw349	bw350
bw351	bw352	bw353
bw354	bw355	bw356
bw357	bw358	bw359
bw360	bw361	bw362
bw363	bw364	bw365
bw366	bw367	bw368
bw369	bw370	bw371
bw372	bw373	bw374
bw375	bw376	bw377
bw378	bw379	bw380
bw381	bw382	bw383
bw384	bw385	bw386
bw387	bw388	bw389
bw390	bw391	bw392
bw393	bw394	bw395
bw396	bw397	bw398
bw399	bw400	bw401
bw402	bw403	bw404
bw405	bw406	bw407
bw408	bw409	bw410
bw411	bw412	bw413
bw414	bw415	bw416
bw417	bw418	bw419
bw420	bw421	bw422
bw423	bw424	bw425
bw426	bw427	bw428
bw429	bw430	bw431
bw432	bw433	bw434
bw435	bw436	bw437
bw438	bw439	bw440
bw441	bw442	bw443
bw444	bw445	bw446
bw447	bw448	bw449
bw450	bw451	bw452
bw453	bw454	bw455
bw456	bw457	bw458
bw459	bw460	bw461
bw462	bw463	bw464
bw465	bw466	bw467
bw468	bw469	bw470
bw471	bw472	bw473
bw474	bw475	bw476
bw477	bw478	bw479
bw480	bw481	bw482
bw483	bw484	bw485
bw486	bw487	bw488
bw489	bw490	bw491
bw492	bw493	bw494
bw495	bw496	bw497
bw498	bw499	bw500
bw501	bw502	bw503
bw504	bw505	bw506
bw507	bw508	bw509
bw510	bw511	bw512
bw513	bw514	bw515
bw516	bw517	bw518
bw519	bw520	bw521
bw522	bw523	bw524
bw525	bw526	bw527
bw528	bw529	bw530
bw531	bw532	bw533
bw534	bw535	bw536
bw537	bw538	bw539
bw540	bw541	bw542
bw543	bw544	bw545
bw546	bw547	bw548
bw549	bw550	bw551
bw552	bw553	bw554
bw555	bw556	bw557
bw558	bw559	bw560
bw561	bw562	bw563
bw564	bw565	bw566
bw567	bw568	bw569
bw570	bw571	bw572
bw573	bw574	bw575
bw576	bw577	bw578
bw579	bw580	bw581
bw582	bw583	bw584
bw585	bw586	bw587
bw588	bw589	bw590
bw591	bw592	bw593
bw594	bw595	bw596
bw597	bw598	bw599
bw600	bw601	bw602
bw603	bw604	bw605
bw606	bw607	bw608
bw609	bw610	bw611
bw612	bw613	bw614
bw615	bw616	bw617
bw618	bw619	bw620
bw621	bw622	bw623
bw624	bw625	bw626
bw627	bw628	bw629
bw630	bw631	bw632
bw633	bw634	bw635
bw636	bw637	bw638
bw639	bw640	bw641
bw642	bw643	bw644
bw645	bw646	bw647
bw648	bw649	bw650
bw651	bw652	bw653
bw654	bw655	bw656
bw657	bw658	bw659
bw660	bw661	bw662
bw663	bw664	bw665
bw666	bw667	bw668
bw669	bw670	bw671
bw672	bw673	bw674
bw675	bw676	bw677
bw678	bw679	bw680
bw681	bw682	bw683
bw684	bw685	bw686
bw687	bw688	bw689
bw690	bw691	bw692
bw693	bw694	bw695
bw696	bw697	bw698
bw699	bw700	bw701
bw702	bw703	bw704
bw705	bw706	bw707
bw708	bw709	bw710
bw711	bw712	bw713
bw714	bw715	bw716
bw717	bw718	bw719
bw720	bw721	bw722
bw723	bw724	bw725
bw726	bw727	bw728
bw729	bw730	bw731
bw732	bw733	bw734
bw735	bw736	bw737
bw738	bw739	bw740
bw741	bw742	bw743
bw744	bw745	bw746
bw747	bw748	bw749
bw750	bw751	bw752
bw753	bw754	bw755
bw756	bw757	bw758
bw759	bw760	bw761
bw762	bw763	bw764
bw765	bw766	bw767
bw768	bw769	bw770
bw771	bw772	bw773
bw774	bw775	bw776
bw777	bw778	bw779
bw780	bw781	bw782
bw783	bw784	bw785
bw786	bw787	bw788
bw789	bw790	bw791
bw792	bw793	bw794
bw795	bw796	bw797
bw798	bw799	bw800
bw801	bw802	bw803
bw804	bw805	bw806
bw807	bw808	bw809
bw810	bw811	bw812
bw813	bw814	bw815
bw816	bw817	bw818
bw819	bw820	bw821
bw822	bw823	bw824
bw825	bw826	bw827
bw828	bw829	bw830
bw831	bw832	bw833
bw834	bw835	bw836
bw837	bw838	bw839
bw840	bw841	bw842
bw843	bw844	bw845
bw846	bw847	bw848
bw849	bw850	bw851
bw852	bw853	bw854
bw855	bw856	bw857
bw858	bw859	bw860
bw861	bw862	bw863
bw864	bw865	bw866
bw867	bw868	bw869
bw870	bw871	bw872
bw873	bw874	bw875
bw876	bw877	bw878
bw879	bw880	bw881
bw882	bw883	bw884
bw885	bw886	bw887
bw888	bw889	bw890
bw891	bw892	bw893
bw894	bw895	bw896
bw897	bw898	bw899
bw900	bw901	bw902
bw903	bw904	bw905
bw906	bw907	bw908
bw909	bw910	bw911
bw912	bw913	bw914
bw915	bw916	bw917
bw918	bw919	bw920
bw921	bw922	bw923
bw924	bw925	bw926
bw927	bw928	bw929
bw930	bw931	bw932
bw933	bw934	bw935
bw936	bw937	bw938
bw939	bw940	bw941
bw942	bw943	bw944
bw945	bw946	bw947
bw948	bw949	bw950
bw951	bw952	bw953
bw954	bw955	bw956
bw957	bw958	bw959
bw960	bw961	bw962
bw963	bw964	bw965
bw966	bw967	bw968
bw969	bw970	bw971
bw972	bw973	bw974
bw975	bw976	bw977
bw978	bw979	bw980
bw981	bw982	bw983
bw984	bw985	bw986
bw987	bw988	bw989
bw990	bw991	bw992
bw993	bw994	bw995
bw996	bw997	bw998
bw999	bw1000	bw1001
bw1002	bw1003	bw1004
bw1005	bw1006	bw1007
bw1008	bw1009	bw1010
bw1011	bw1012	bw1013
bw1014	bw1015	bw1016
bw1017	bw1018	bw1019
bw1020	bw1021	bw1022
bw1023	bw1024	bw1025
bw1026	bw1027	bw1028
bw1029	bw1030	bw1031
bw1032	bw1033	bw1034
bw1035	bw1036	bw1037
bw1038	bw1039	bw1040
bw1041	bw1042	bw1043
bw1044	bw1045	bw1046
bw1047	bw1048	bw1049
bw1050	bw1051	bw1052
bw1053	bw1054	bw1055
bw1056	bw1057	bw1058
bw1059	bw1060	bw1061
bw1062	bw1063	bw1064
bw1065	bw1066	bw1067
bw1068	bw1069	bw1070
bw1071	bw1072	bw1073
bw1074	bw1075	bw1076
bw1077	bw1078	bw1079
bw1080	bw1081	bw1082
bw1083	bw1084	bw1085
bw1086	bw1087	bw1088
bw1089	bw1090	bw1091
bw1092	bw1093	bw1094

Pict = p1
 Real = -0.76000000000000
 Imag = 0.01000000000000
 Side = 0.02000000000000
 Size = 450

Current ColorMap = bw128

Canvas Real = -0.76000000000000
 Canvas Imag = 0.01000000000000
 Canvas Side = 0.02000000000000

Bone

Load	Disp	Inv.	Note	Quit
p000	p10	p20	p1	p2
p4	p28	p12	p31	p29
p25	p11	p34	p13	p33
p29	p32	p15	p14	p17
p16				

Red Green Blue

Rotate Speed:

1/5 Rotate

Default

1 Pixel

1/5 Rotate

Full Rotate

Invert

Map Rotate

ColorMaps

bw000	bw001	bw002
bw004	bw016	bw032
bw128	cos01	cos02
cos04	cos16	lin01
lin02	lin03	lin16
stp01	stp02	stp04
stp16		

Pict = p20
Real = -0.63235290000000
Imag = -0.70000000000000
Size = 850
Size = 850

Current ColorMap = hv128

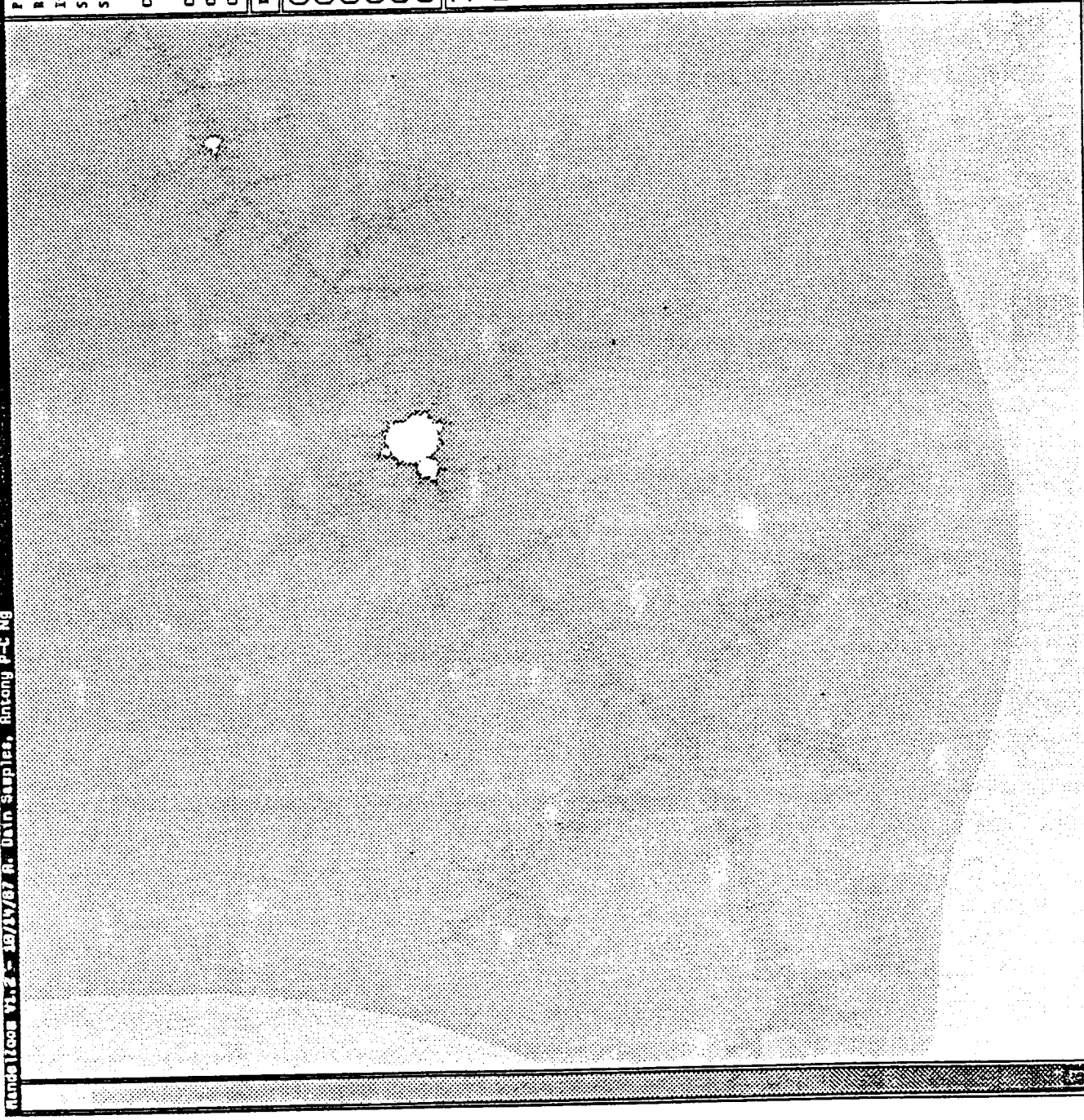
Canvas Real = -0.63235290000000
Canvas Imag = -0.70000000000000
Canvas Size = 0.05882353000000

Done			
Load	Disp	Inv.	Note
p000	p10	p20	p1
p4	p24	p12	p31
p25	p11	p34	p13
p29	p32	p15	p14
p16			p17

Rotate Speed:
[Slider]

Default
1/5 Rotate
Full Rotate
Invert
Map Rotate

ColorMaps	
hv000	hv001
hv004	hv016
hv128	cos01
cos04	cos16
lin02	lin03
stip01	stip02
stip16	



Pict - p33
Real - -1.2554225605364
Imag - 0.02712537024222
Side - 0.00020905190311
Size - 850

Current ColorMap - b0128

Canvas Real - -1.2554225605364
Canvas Imag - 0.02712537024222
Canvas Side - 0.00020905190311

Base

Load	Disp	Inv	Note	Quit
p000	p10	p20	p1	p2
p1	p28	p12	p31	p30
p25	p11	p34	p13	p22
p29	p32	p15	p14	p17
p16				

Rotate Speed:

Default

1/5 Rotate

Full Rotate

Invert

Map Rotate

Ctrl + Caps

b000	b001	b002
b004	b016	b032
b0128	cos0	cos02
cos04	cos16	lin01
lin02	lin03	lin16
slp01	slp02	slp04
slp16		